

Differentiable Simulators: Implications for Software Development and Applications to Reduced-Order and Data-Driven Modelling

Knut-Andreas Lie

SINTEF Digital, Oslo, Norway

SIAM Conference on Mathematical & Computational Issues in the Geosciences (GS23)
June 19–22, 2023, Bergen, Norway

Differentiable programming is a programming paradigm in which a numeric computer program can be differentiated throughout via automatic differentiation.

From: wikipedia.org/wiki/Differentiable_programming

Differentiable programming is a programming paradigm in which a numeric computer program can be differentiated throughout via automatic differentiation.

From: [wikipedia.org/wiki/Differentiable_programming](https://en.wikipedia.org/wiki/Differentiable_programming)

- provides gradients that indicate how input parameters should change to produce a change in the output state(s)
- enables gradient-based optimization, allowing for efficient parameter tuning and optimization of models
- leveraged in frameworks like TensorFlow, Theano, and PyTorch to enhance model training and optimization
- can be used to develop differentiable PDE-based simulators

Differentiable programming is a programming paradigm in which a numeric computer program can be differentiated throughout via automatic differentiation.

From: wikipedia.org/wiki/Differentiable_programming

- provides gradients that indicate how input parameters should change to produce a change in the output state(s)
- enables gradient-based optimization, allowing for efficient parameter tuning and optimization of models
- leveraged in frameworks like TensorFlow, Theano, and PyTorch to enhance model training and optimization
- can be used to develop differentiable PDE-based simulators

Two types of approaches

- **static, compiled graph-based approaches** like TensorFlow and Theano
- good computer optimization and scaling
- static nature limits interactivity and type of program
- **operator overloading, dynamic graph-based approaches** like PyTorch and Zygote (Julia)
- dynamic/interactive use, more flexible programming
- interpreter overhead, less optimization

Either case: rely on **computational graphs**

Typically has support for GPU, TPU, or other accelerators

Automatic differentiation is a set of techniques to compute the derivative or gradient of a function with respect to its inputs

Basic concepts:

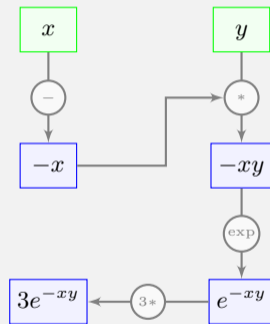
- can leverage the chain rule of calculus to break down complex computations into a sequence of elementary operations
- these operations consist of a limited set of arithmetic operations and elementary functions
- each elementary operation has known differential rules
- by evaluating the derivatives of each elementary operation, one obtains exact derivatives at given input values

Automatic differentiation is a set of techniques to compute the derivative or gradient of a function with respect to its inputs

Basic concepts:

- can leverage the chain rule of calculus to break down complex computations into a sequence of elementary operations
- these operations consist of a limited set of arithmetic operations and elementary functions
- each elementary operation has known differential rules
- by evaluating the derivatives of each elementary operation, one obtains exact derivatives at given input values

Example: evaluate $3 * \exp(-x * y)$



Forward mode:

- computes both function values and their derivatives by propagating computations forward through the computational graph, starting from the input variables
- well-suited for functions with a small number of inputs and a large number of outputs

Reverse/backward mode:

- starts from the output variables, works backward through the computational graph, accumulating sensitivities or gradients with respect to the inputs
- is efficient for large number of inputs compared to outputs, making it particularly useful for gradient-based optimization

Many libraries also use source code transformation. Mixed-mode versions may also be needed

PDE-based simulators have approximately as many outputs as inputs, and question is how to best exploit inherent sparsity structure

Software

Description



A unique research and prototyping tool used all over the world, e.g., as evidenced by more than 210 master/doctoral theses and 600 external journal and proceedings papers

URL: mrst.no



Open-source reservoir simulator aimed at commercial application based on industry-standard black-oil type models. Used by Equinor for asset models on the Norwegian Continental Shelf.

URL: opm-project.org

Sandve, MS35, Wed 09:10, Plenary Hall



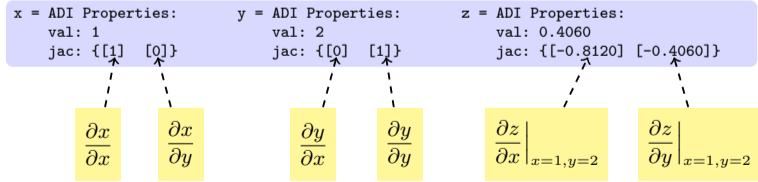
Experimental Julia framework for fully differentiable multi-physics simulators. Extensive functionality for reservoir simulation (JutulDarcy.jl) and computational electrochemistry (BattMo.jl)

URL: github.com/sintefmath/Jutul.jl

Møyner, MS35, Wed 10:25, Plenary Hall

- Introduce an extended pair, $\langle v, v_x \rangle$, to represent the value v and its derivative v_x to a given input variable $x = \langle x, 1 \rangle$
- Use chain rule and elementary derivative rules to mechanically accumulate derivatives at *specific values* of x
 - Elementary: $v = \sin(x) \longrightarrow \langle v \rangle = \langle \sin x, \cos x \rangle$
 - Arithmetic: $v = f * g \longrightarrow \langle v \rangle = \langle f * g, f * g_x + f_x * g \rangle$
 - Chain rule: $v = \exp(f(x)) \longrightarrow \langle v \rangle = \langle \exp(f(x)), \exp(f(x))f'(x) \rangle$
- Use operator overloading to avoid messing up code

```
[x,y] = initVariablesADI(1,2);
z = 3*exp(-x*y)
```



- **Accuracy:** exact derivatives, unlike for finite-difference approximations
- **Correctness:** avoids error-prone manual derivation and implementation
- **Efficiency:** avoids time-consuming manual derivation and implementation
- **Flexibility:** enables users to focus on the model's logic rather than manual derivation of derivatives
- **Adaptability:** simple differentiation of underlying equations with respect to parameters or variables enables sensitivity analysis, optimization, and parameter estimation

- **Accuracy:** exact derivatives, unlike for finite-difference approximations
- **Correctness:** avoids error-prone manual derivation and implementation
- **Efficiency:** avoids time-consuming manual derivation and implementation
- **Flexibility:** enables users to focus on the model's logic rather than manual derivation of derivatives
- **Adaptability:** simple differentiation of underlying equations with respect to parameters or variables enables sensitivity analysis, optimization, and parameter estimation

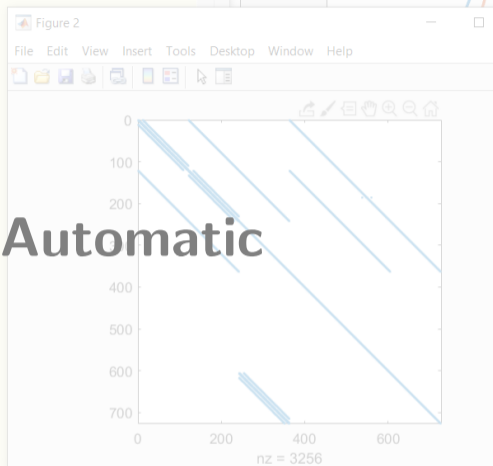
However, there are potential pitfalls

- May struggle with complex control flow such as loops, conditionals, recursions, linear solvers, etc.
- Increased memory consumption (because of intermediate storage)
- Computational overhead: all intermediate steps computed if straightforward implementation, compiler must perform simplification of expressions in advanced versions
- Primarily designed for continuous functions/variables
- Can be misleading if you fail to specify independent variables correctly

Rapid Prototyping Using Automatic Differentiation

AD Applied to Dynamic Variables

```
% Pressure differences across each interface
146 dp = grad(p);
147 dpW = dp-g*avg(rW).*gradz;
148 dpO = dp-g*avg(rO).*gradz;
149
150
151 % Phase fluxes:
152 % Density and mobility is taken upwinded (value on interface is
153 % defined as taken from the cell from which the phase flow is
154 % currently coming from). This gives more physical solutions than
155 % averaging or downwinding.
156 vW = -upw(value(dpW) <= 0, rW.*mobW).*T.*dpW;
157 vO = -upw(value(dpO) <= 0, rO.*mobO).*T.*dpO;
158
159 % Conservation of water and oil
160 water = (1/dt(n)).*(vol.*rW.*sW - vol0.*rW0.*sW0) + div(vW);
161 oil = (1/dt(n)).*(vol.*rO.*sO - vol0.*rO0.*sO0) + div(vO);
162
163 % Injector: volumetric source term multiplied by surface density
164 water(inIx) = water(inIx) - inRate.*rhoWS;
165
166 % Producer: replace equations by new ones specifying fixed pressure
167 % and zero water saturation
168 oil(outIx) = p(outIx) - outPres;
169 water(outIx) = sW(outIx);
170
171 % Collect and concatenate all equations (i.e., assemble and
172 % linearize system)
173 eqs = {oil, water};
174 eq = combineEquations(eqs{:});
175
176 % Measure condition number
177 cnd(i) = condest(eq.jac[1]); i = i+1;
178
179 % Compute Newton update and update variable
180 res = eq.res;
```



[ADI](#) with properties:

```
val: [726x1 double]
jac: [[726x726 double]]
```

`fx K>>`

`%#ok<SAGROW>`

Imagine you had discrete differentiation operators you could use like their continuous counterparts when writing code:

Continuous equations:

$$\nabla \cdot (\mathbf{K} \nabla p) + q = 0$$

Discrete equations:

$$\text{div}(\mathbf{K} \text{grad}(p)) + q = 0$$

To explain how to achieve this, we write the equation in first-order form instead

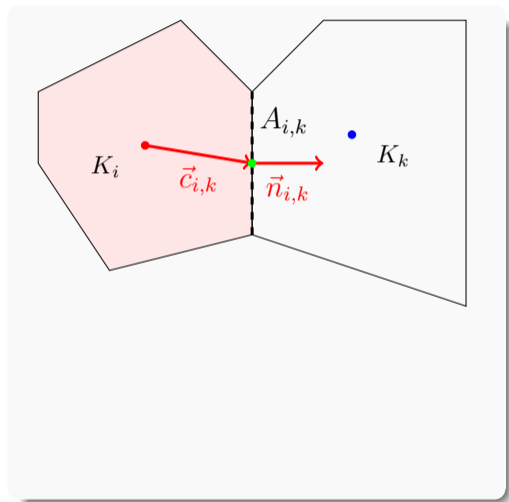
$$-\nabla \cdot \vec{v} + q = 0, \quad \vec{v} = -K \nabla p$$

Conservation of mass:

$$\sum_{\Gamma_f \in \partial\Omega_c} \int_{\Gamma_f} \vec{v} \cdot \vec{n} \, ds = \int_{\Omega_c} \nabla \cdot \vec{v} \, d\vec{x} = \int_{\Omega_c} q \, d\vec{x}$$

discrete:

$$\operatorname{div}(\mathbf{v})[c] = \mathbf{q}[c]$$



Conservation of mass:

$$\sum_{\Gamma_f \in \partial\Omega_c} \int_{\Gamma_f} \vec{v} \cdot \vec{n} ds = \int_{\Omega_c} \nabla \cdot \vec{v} d\vec{x} = \int_{\Omega_c} q d\vec{x}$$

discrete:

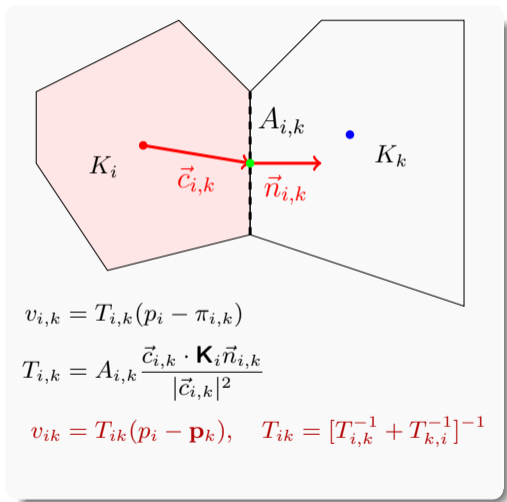
$$\operatorname{div}(\mathbf{v})[c] = \mathbf{q}[c]$$

Darcy's law:

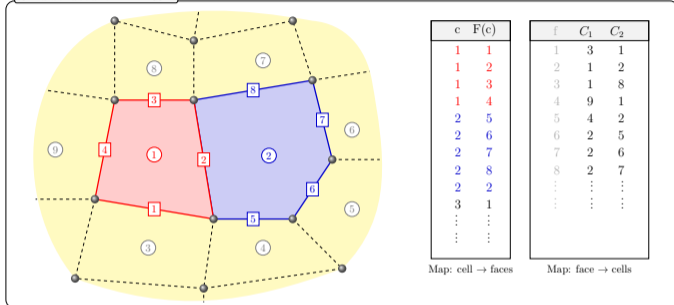
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f ds = - \int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f ds$$

discrete:

$$\mathbf{v}[f] = -\mathbf{T}[f] \operatorname{grad}(\mathbf{p})[f]$$



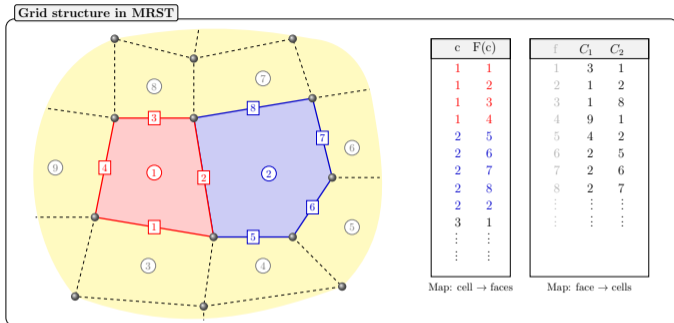
Grid structure in MRST



The discrete grad operator maps from cell pair $C_1(f), C_2(f)$ to face f :

$$\text{grad}(\mathbf{p})[f] = \mathbf{p}[C_2(f)] - \mathbf{p}[C_1(f)],$$

$\mathbf{p}[c]$: scalar quantity associated with cell c



The discrete grad operator maps from cell pair $C_1(f), C_2(f)$ to face f :

$$\text{grad}(\mathbf{p})[f] = \mathbf{p}[C_2(f)] - \mathbf{p}[C_1(f)],$$

$\mathbf{p}[c]$: scalar quantity associated with cell c

Introduce sparse matrix:

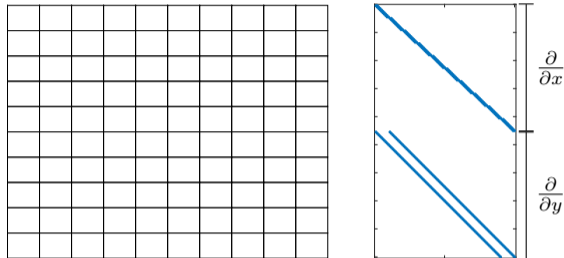
$$D_{ij} = \begin{cases} 1, & j = C_1(i), \\ -1, & j = C_2(i), \\ 0, & \text{otherwise} \end{cases}$$

so that

$$\text{grad}(\mathbf{x}) = -\mathbf{D}\mathbf{x}$$

and likewise,

$$\text{div}(\mathbf{y}) = \mathbf{D}^T \mathbf{y}$$



The discrete grad operator maps from cell pair $C_1(f), C_2(f)$ to face f :

$$\mathbf{grad}(\mathbf{p})[f] = \mathbf{p}[C_2(f)] - \mathbf{p}[C_1(f)],$$

$\mathbf{p}[c]$: scalar quantity associated with cell c

Introduce sparse matrix:

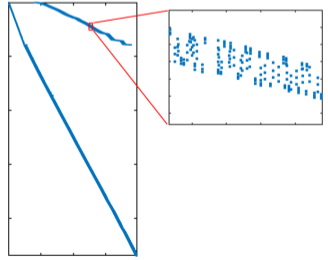
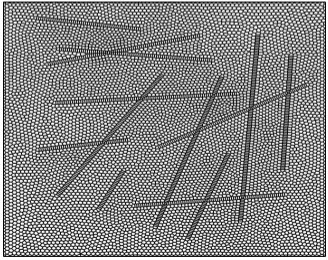
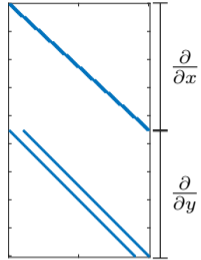
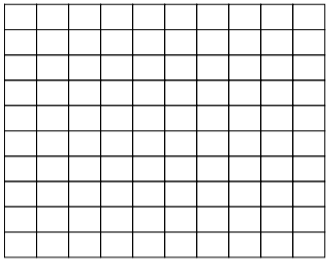
$$D_{ij} = \begin{cases} 1, & j = C_1(i), \\ -1, & j = C_2(i), \\ 0, & \text{otherwise} \end{cases}$$

so that

$$\mathbf{grad}(\mathbf{x}) = -\mathbf{D}\mathbf{x}$$

and likewise,

$$\mathbf{div}(\mathbf{y}) = \mathbf{D}^T \mathbf{y}$$



The discrete grad operator maps from cell pair $C_1(f), C_2(f)$ to face f :

$$\text{grad}(\mathbf{p})[f] = \mathbf{p}[C_2(f)] - \mathbf{p}[C_1(f)],$$

$\mathbf{p}[c]$: scalar quantity associated with cell c

Introduce sparse matrix:

$$D_{ij} = \begin{cases} 1, & j = C_1(i), \\ -1, & j = C_2(i), \\ 0, & \text{otherwise} \end{cases}$$

so that

$$\text{grad}(\mathbf{x}) = -\mathbf{D}\mathbf{x}$$

and likewise,

$$\text{div}(\mathbf{y}) = \mathbf{D}^T \mathbf{y}$$

Continuous

Incompressible flow:

$$\nabla \cdot (\mathbf{K}\nabla p) + q = 0$$

Compressible flow:

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla \cdot (\rho\mathbf{K}\nabla p) + q = 0$$

Discrete in MATLAB

Incompressible flow:

```
eq = div(T .* grad(p)) + q;
```

Compressible flow:

```
eq = (pv(p).*rho(p)-pv(p0).*rho(p0))/dt ...  
+ div(avg(rho(p)).*T.*grad(p))+q;
```

Continuous

Incompressible flow:

$$\nabla \cdot (\mathbf{K} \nabla p) + q = 0$$

Compressible flow:

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla \cdot (\rho\mathbf{K}\nabla p) + q = 0$$

Discrete in MATLAB

Incompressible flow:

$$\text{eq} = \text{div}(T .* \text{grad}(p)) + q;$$

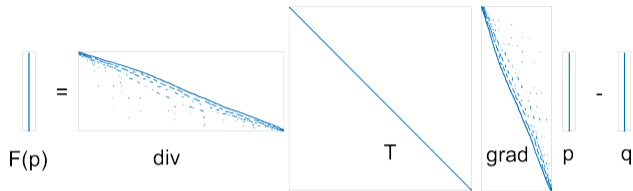
Compressible flow:

$$\text{eq} = (\text{pv}(p) .* \text{rho}(p) - \text{pv}(p0) .* \text{rho}(p0)) / \text{dt} \dots \\ + \text{div}(\text{avg}(\text{rho}(p)) .* T .* \text{grad}(p)) + q;$$

Discretization of flow models leads to large systems of nonlinear algebraic equations, typically linearized and solved with Newton's method

$$\mathbf{F}(\mathbf{u}) = \mathbf{0} \quad \Rightarrow \quad \frac{\partial \mathbf{F}}{\partial \mathbf{u}}(\mathbf{u}^i)(\mathbf{u}^{i+1} - \mathbf{u}^i) = -\mathbf{F}(\mathbf{u}^i)$$

Discretization gives the residual flow equation



```
% Grid and petrophysics
```

```
load seamount
G = pebi(triangleGrid([x(:) y(:)]));
G = computeGeometry(G);
rock = makeRock(G, 1, 1);
nc = G.cells.num;
```

```
% Discrete operators
```

```
S = setupOperatorsTPFA(G,rock);
```

```
% Unknown p (AD) + source and sink
```

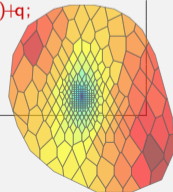
```
p = initVariablesADI(zeros(nc,1));
q = zeros(nc, 1);
q([135 282 17]) = [-1 .5 .5];
```

```
% Evaluate residual equation
```

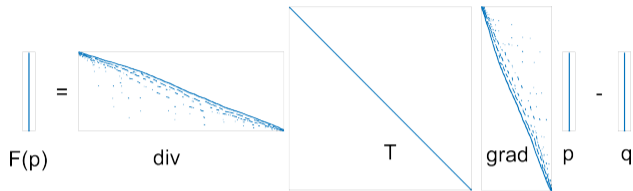
```
eq = S.Div(S.T.*S.Grad(p))+q;
eq(1) = eq(1) + p(1);
```

```
% Solve linear system
```

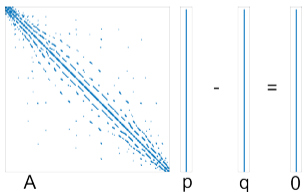
```
p = -eq.jac{1}\eq.val;
```



Discretization gives the residual flow equation



Setting the residual to zero, gives a linear system



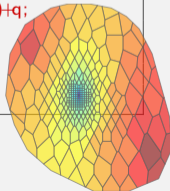
```
% Grid and petrophysics
load seamount
G = pebi(triangleGrid([x(:) y(:)]));
G = computeGeometry(G);
rock = makeRock(G, 1, 1);
nc = G.cells.num;
```

```
% Discrete operators
S = setupOperatorsTPFA(G,rock);
```

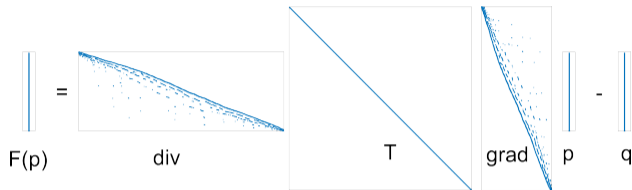
```
% Unknown p (AD) + source and sink
p = initVariablesADI(zeros(nc,1));
q = zeros(nc, 1);
q([135 282 17]) = [-1 .5 .5];
```

```
% Evaluate residual equation
eq = S.Div(S.T.*S.Grad(p))+q;
eq(1) = eq(1) + p(1);
```

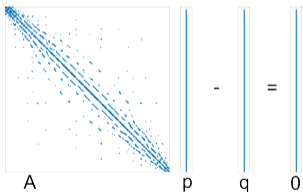
```
% Solve linear system
p = -eq.jac{1}\eq.val;
```



Discretization gives the residual flow equation



Setting the residual to zero, gives a linear system



With AD, we can assemble the linear system implicitly and solve it with Newton's method since $\partial \mathbf{F} / \partial \mathbf{p} = \mathbf{A}$

```

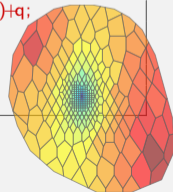
% Grid and petrophysics
load seamount
G = pebi(triangleGrid([x(:) y(:)]));
G = computeGeometry(G);
rock = makeRock(G, 1, 1);
nc = G.cells.num;

% Discrete operators
S = setupOperatorsTPFA(G,rock);

% Unknown p (AD) + source and sink
p = initVariablesADI(zeros(nc,1));
q = zeros(nc, 1);
q([135 282 17]) = [-1 .5 .5];

% Evaluate residual equation
eq = S.Div(S.T.*S.Grad(p))+q;
eq(1) = eq(1) + p(1);

% Solve linear system
p = -eq.jac{1}\eq.val;
    
```



Discretization gives the residual flow equation



Not much gained so far. However, the same idea applies to more complex nonlinear models

- You implement the residual equations $\mathbf{R}(\mathbf{x}_{n+1}, \mathbf{x}_n; \mathbf{p}) = \mathbf{0}$
- The AD library linearizes and assembles Jacobians, $\partial \mathbf{R} / \partial \mathbf{x}$
- The nonlinear system of algebraic equations can be solved with Newton's method

$$-\mathbf{J}\Delta \mathbf{x} = \mathbf{R}(\mathbf{x}), \quad \mathbf{J} = \frac{\partial \mathbf{R}}{\partial \mathbf{x}}$$



With AD, we can assemble the linear system implicitly and solve it with Newton's method since $\partial \mathbf{F} / \partial \mathbf{p} = \mathbf{A}$

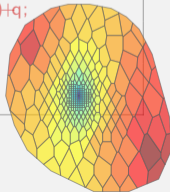
```
% Grid and petrophysics
load seamount
G = pebi(triangleGrid([x(:) y(:)]));
G = computeGeometry(G);
rock = makeRock(G, 1, 1);
nc = G.cells.num;

% Discrete operators
S = setupOperatorsTPFA(G,rock);

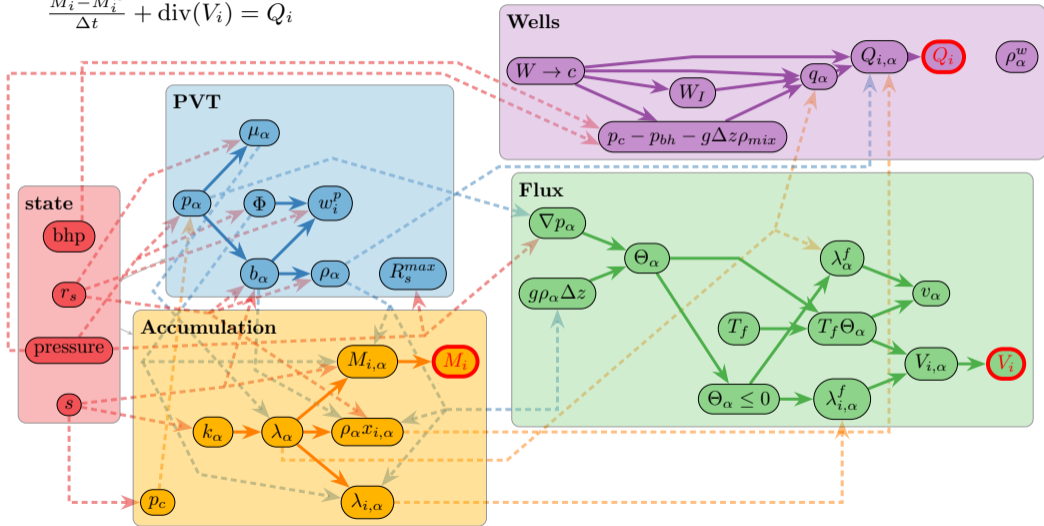
% Unknown p (AD) + source and sink
p = initVariablesADI(zeros(nc,1));
q = zeros(nc, 1);
q([135 282 17]) = [-1 .5 .5];

% Evaluate residual equation
eq = S.Div(S.T.*S.Grad(p))+q;
eq(1) = eq(1) + p(1);

% Solve linear system
p = -eq.jac{1}\eq.val;
```



$$\frac{M_i - M_i^n}{\Delta t} + \text{div}(V_i) = Q_i$$



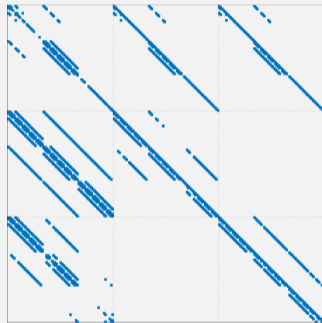
Example: black-oil model

Naive implementation of AD is powerful, but can be slow compared to manual codes

MRST:

- vectorized AD library with variable-major ordering
- acceleration through C++-extensions

Example: 3-phase flow in 3D



Variable-major ordering

Naive implementation of AD is powerful, but can be slow compared to manual codes

MRST:

- vectorized AD library with variable-major ordering
- acceleration through C++-extensions

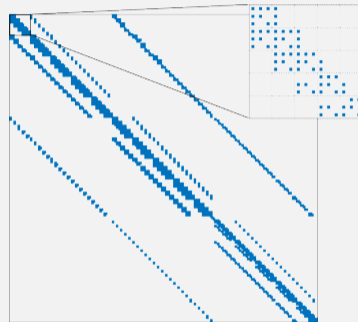
OPM:

- switch from variable-major to cell-major gave speedup
- further speedup from TPFA-specialized AD library

Jutul: both version available

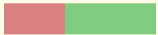


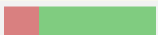

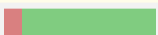

Many tricks generally necessary for high efficiency

Example: 3-phase flow in 3D



Cell-major ordering

Performance using Jutul (with Hypr linear solver on a single thread)

Name	Model	#cells	#wells	AD : linear solver
SPE1	black-oil, 3-phase	300	2	 40%
Fractures	2-phase, 3-component (CO ₂)	7 932	2	 9.4%
SPE9	black-oil, 3-phase	9 000	26	 28%
Egg	water-oil, 2-phase	18 553	12	 23%
Olympus	water-oil, 2-phase	192 749	18	 14%
SPE10	water-oil, 2-phase	1 122 000	5	 12%
Sleipner	water-gas (CO ₂), 2-phase	1 986 176	1	 23%

For OPM Flow, the AD and property fraction is typically 30% for models of commercial interest across a wide range of parallelism

Two physical models

Model A with linearization

$$\mathbf{R}_a(\mathbf{x}_a) = \mathbf{0}, \quad -\mathbf{J}_{aa}\Delta\mathbf{x}_a = -\mathbf{R}_a$$

Model B with linearization

$$\mathbf{R}_b(\mathbf{x}_b) = \mathbf{0}, \quad -\mathbf{J}_{bb}\Delta\mathbf{x}_b = -\mathbf{R}_b$$

Combined model

$$R(\mathbf{x}_a, \mathbf{x}_b) = \begin{bmatrix} R_a(\mathbf{x}_a, \mathbf{x}_b) \\ R_b(\mathbf{x}_b, \mathbf{x}_a) \end{bmatrix} = \mathbf{0}$$

$$\begin{bmatrix} \mathbf{J}_{aa} & \mathbf{J}_{ab} \\ \mathbf{J}_{ba} & \mathbf{J}_{bb} \end{bmatrix} \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{bmatrix} = \begin{bmatrix} \mathbf{R}_a \\ \mathbf{R}_b \end{bmatrix}$$

Model equations, isothermal case:

$$\frac{\partial}{\partial t} [\phi \rho(p)] + \nabla \cdot [\rho(p) \vec{v}] = q, \quad \vec{v} = -\frac{\mathbf{K}}{\mu(p)} [\nabla p - g \rho(p) \nabla z],$$

Constitutive laws and operators

```
pvr = poreVolume(G, rock);
pv  = @(p) pvr .* exp( cr * (p - pr) );
:
rho  = @(p) rhor.*(1+(cp*(p - pr)));
mu   = @(p) mu0*(1+cmup*(p-p_r));
```

Discrete equations

```
v = @(p) -(Tr./mu(avg(p))) ...
        .*( grad(p) - g*avg(rho(p)).*dz );

pEq = @(p, p0, dt) ...
      (1/dt)*(pv(p).*rho(p) - pv(p0).*rho(p0)) ...
      + div( avg(rho(p)).*v(p) );
```

Model equations, thermal case:

$$\frac{\partial}{\partial t} [\phi \rho(p, T)] + \nabla \cdot [\rho(p, T) \vec{v}] = q, \quad \vec{v} = -\frac{\mathbf{K}}{\mu(p, T)} [\nabla p - g \rho(p, T) \nabla z],$$

$$\frac{\partial}{\partial t} [\phi(p, T) \rho E_f(p, T) + (1 - \phi) E_r(p, T)] + \nabla \cdot [(\rho H_f \vec{v})(p, T)] - \nabla \cdot [\kappa \nabla T] = q_e.$$

Constitutive laws and operators

```
pvr = poreVolume(G, rock);
pv  = @(p) pvr .* exp( cr * (p - pr) );
spv = @(p) G.cells.volumes - pv(p);
:
rho  = @(p,T) rhor.*(1+(cp*(p - pr))).*exp(-ct*(T-Tr));
mu   = @(p,T) mu0*(1+cmup*(p - pr)).*exp(-cmut*(T-Tr));
:
Hf   = @(p,T) Cw*T + (1-Tr*ct).*(p-pr)./rho(p,T);
Ef   = @(p,T) Hf(p,T) - p./rho(p,T);
Er   = @(T) Cr*T;
```

Discrete equations

```
v = @(p,T) -(Tr./mu(avg(p),avg(T))) ...
           .*( grad(p) - g*avg(rho(p,T)).*dz );

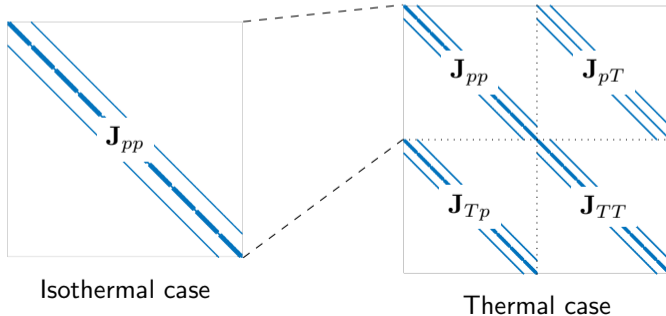
pEq = @(p,T, p0, T0, dt) ...
      (1/dt)*(pv(p).*rho(p,T) - pv(p0).*rho(p0,T0)) ...
      + div( avg(rho(p,T)).*v(p,T) );

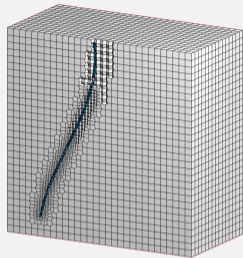
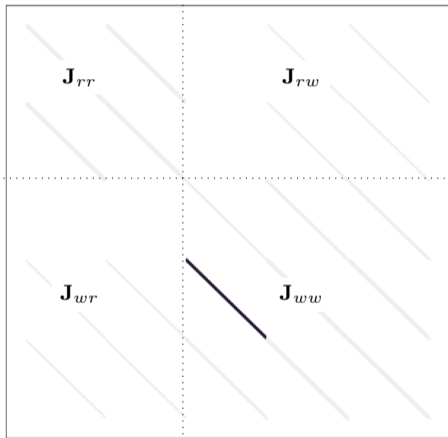
hEq = @(p, T, p0, T0, dt) ...
      (1/dt)*( pv(p).*rho(p,T).*Ef(p,T) + spv(p).*Er(T)
              - pv(p0).*rho(p0,T0).*Ef(p0,T0) - spv(p0).*Er(T0) ) ...
      + div( upw(Hf(p,T),v(p,T)>0).*avg(rho(p,T)).*v(p,T) ) ...
      + div( -Th.*grad(T) );
```


Model equations, thermal case:

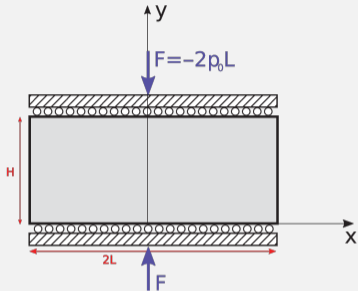
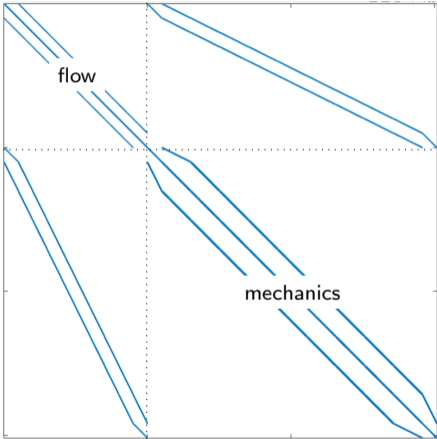
$$\frac{\partial}{\partial t} [\phi \rho(p, T)] + \nabla \cdot [\rho(p, T) \vec{v}] = q, \quad \vec{v} = -\frac{\mathbf{K}}{\mu(p, T)} [\nabla p - g \rho(p, T) \nabla z],$$

$$\frac{\partial}{\partial t} [\phi(p, T) \rho E_f(p, T) + (1 - \phi) E_r(p, T)] + \nabla \cdot [(\rho H_f \vec{v})(p, T)] - \nabla \cdot [\kappa \nabla T] = q_e.$$



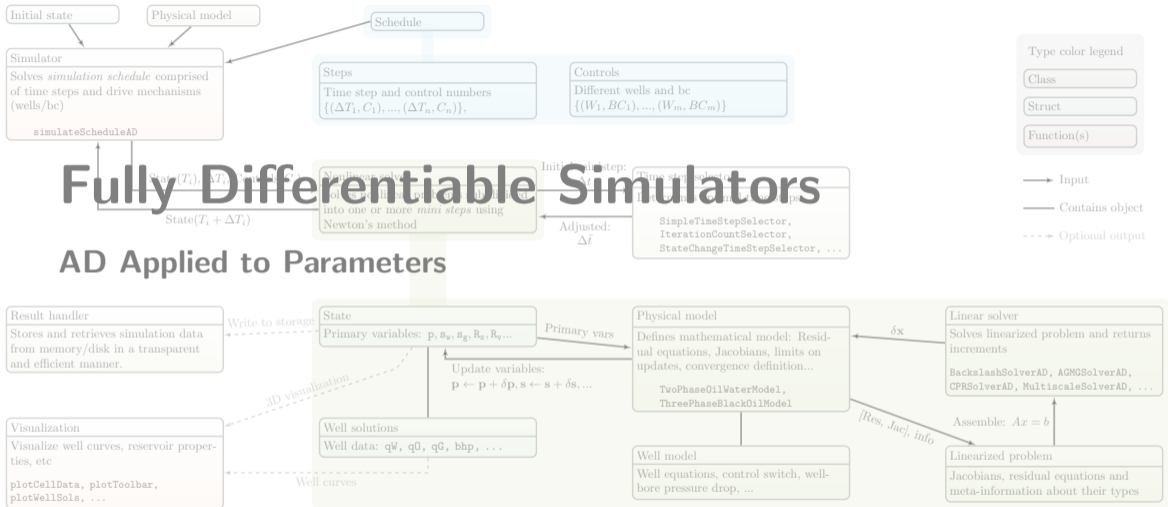


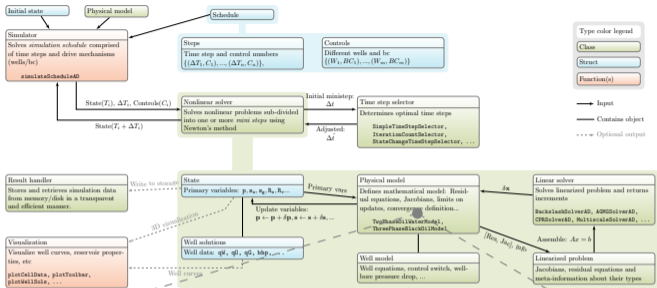
- Model A: standard reservoir model
- Model B: multisegment well model, representing pressure drops and well-bore storage (often: drift flux)



Mandel's problem:

- famous problem from poroelasticity
- demonstrates two-way coupling of fluid pressure and mechanical deformation





Stacking the residual equations, we can write the simulator as a system

$$S(\mathbf{x}, \mathbf{p}, \mathbf{q}) = \mathbf{0}$$

where

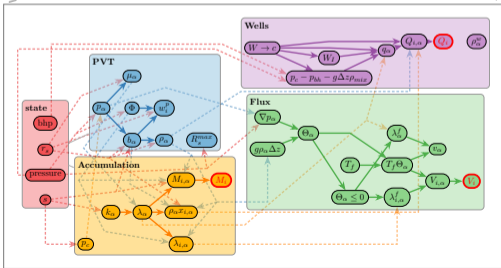
\mathbf{x} : vector of all states in time

\mathbf{p} : parameter vector

\mathbf{q} : vector of driving forces

S cannot easily be differentiated because of complex program control, iterative solvers, etc.

Instead: compute gradients with **adjoint method**
(Similar to backpropagation from machine learning)



Define a Lagrange function (observed quantity penalized by simulator residual)

$$J_{\lambda} = O(\mathbf{x}(\mathbf{p})) + \boldsymbol{\lambda}^{\top} \mathbf{S}(\mathbf{x}(\mathbf{p}), \mathbf{p}, \mathbf{q})$$

Gradient: differentiate with respect to \mathbf{p}

$$\frac{dJ_{\lambda}}{d\mathbf{p}} = \left(\frac{\partial O}{\partial \mathbf{x}} + \boldsymbol{\lambda}^{\top} \frac{\partial \mathbf{S}}{\partial \mathbf{x}} \right) \frac{d\mathbf{x}}{d\mathbf{p}} + \boldsymbol{\lambda}^{\top} \frac{\partial \mathbf{S}}{\partial \mathbf{p}} + \mathbf{S}^{\top} \frac{d\boldsymbol{\lambda}}{d\mathbf{p}}$$

Define a Lagrange function (observed quantity penalized by simulator residual)

$$J_\lambda = O(\mathbf{x}(\mathbf{p})) + \boldsymbol{\lambda}^\top \mathbf{S}(\mathbf{x}(\mathbf{p}), \mathbf{p}, \mathbf{q})$$

Gradient: differentiate with respect to \mathbf{p}

$$\frac{dJ_\lambda}{d\mathbf{p}} = \left(\frac{\partial O}{\partial \mathbf{x}} + \boldsymbol{\lambda}^\top \frac{\partial \mathbf{S}}{\partial \mathbf{x}} \right) \frac{d\mathbf{x}}{d\mathbf{p}} + \boldsymbol{\lambda}^\top \frac{\partial \mathbf{S}}{\partial \mathbf{p}} + \mathbf{S}^\top \frac{d\boldsymbol{\lambda}}{d\mathbf{p}}$$

$= 0$

Forward simulation:

$$\mathbf{S}(\mathbf{x}(\mathbf{p}), \mathbf{p}, \mathbf{q}) = 0$$

Solved with a standard simulator

Define a Lagrange function (observed quantity penalized by simulator residual)

$$J_\lambda = O(\mathbf{x}(\mathbf{p})) + \boldsymbol{\lambda}^\top \mathbf{S}(\mathbf{x}(\mathbf{p}), \mathbf{p}, \mathbf{q})$$

Gradient: differentiate with respect to \mathbf{p}

$$\frac{dJ_\lambda}{d\mathbf{p}} = \left(\frac{\partial O}{\partial \mathbf{x}} + \boldsymbol{\lambda}^\top \frac{\partial \mathbf{S}}{\partial \mathbf{x}} \right) \frac{d\mathbf{x}}{d\mathbf{p}} + \boldsymbol{\lambda}^\top \frac{\partial \mathbf{S}}{\partial \mathbf{p}} + \mathbf{S}^\top \frac{d\boldsymbol{\lambda}}{d\mathbf{p}}$$

Adjoint equations:

$(\partial \mathbf{S} / \partial \mathbf{x})^\top \boldsymbol{\lambda} = -(\partial O / \partial \mathbf{x})^\top$
Solved backward for $\boldsymbol{\lambda}$ after solving forward for \mathbf{x}

Forward simulation:

$\mathbf{S}(\mathbf{x}(\mathbf{p}), \mathbf{p}, \mathbf{q}) = 0$
Solved with a standard simulator

Define a Lagrange function (observed quantity penalized by simulator residual)

$$J_\lambda = O(\mathbf{x}(\mathbf{p})) + \boldsymbol{\lambda}^\top \mathbf{S}(\mathbf{x}(\mathbf{p}), \mathbf{p}, \mathbf{q})$$

Gradient: differentiate with respect to \mathbf{p}

$$\frac{dJ_\lambda}{d\mathbf{p}} = \left(\frac{\partial O}{\partial \mathbf{x}} + \boldsymbol{\lambda}^\top \frac{\partial \mathbf{S}}{\partial \mathbf{x}} \right) \frac{d\mathbf{x}}{d\mathbf{p}} + \boldsymbol{\lambda}^\top \frac{\partial \mathbf{S}}{\partial \mathbf{p}} + \mathbf{S}^\top \frac{d\boldsymbol{\lambda}}{d\mathbf{p}}$$

Adjoint equations:

$(\partial \mathbf{S} / \partial \mathbf{x})^\top \boldsymbol{\lambda} = -(\partial O / \partial \mathbf{x})^\top$
Solved backward for $\boldsymbol{\lambda}$ after solving forward for \mathbf{x}

Automatic differentiation:

$\partial \mathbf{S} / \partial \mathbf{p}$ computed “behind the curtain” by the code during the backward adjoint solve (technically: set \mathbf{p} as independent variable)

Forward simulation:

$\mathbf{S}(\mathbf{x}(\mathbf{p}), \mathbf{p}, \mathbf{q}) = 0$
Solved with a standard simulator

We can write the Lagrange function as a sum over all time steps:

$$J_{\lambda} = \sum_{n=1}^N \left[O_n(\mathbf{x}_n(\mathbf{p})) + \boldsymbol{\lambda}_n^{\top} \mathbf{R}_n(\mathbf{x}_n(\mathbf{p}), \mathbf{x}_{n-1}(\mathbf{p}), \mathbf{p}, \mathbf{q}_n) \right]$$

Differentiate with respect to \mathbf{p}

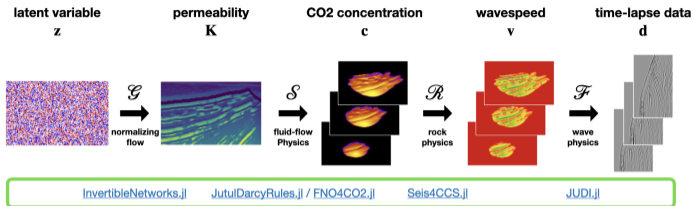
$$\frac{dJ_{\lambda}}{d\mathbf{p}} = \sum_{n=1}^N \left[\left(\frac{\partial O_n}{\partial \mathbf{x}_n} + \boldsymbol{\lambda}_n^{\top} \frac{\partial \mathbf{R}_n}{\partial \mathbf{x}_n} + \boldsymbol{\lambda}_n^{\top} \frac{\partial \mathbf{R}_{n+1}}{\partial \mathbf{x}_n} \right) \frac{d\mathbf{x}_n}{d\mathbf{p}} + \boldsymbol{\lambda}_n^{\top} \frac{\partial \mathbf{R}_n}{\partial \mathbf{p}} + \mathbf{S}_n^{\top} \frac{d\boldsymbol{\lambda}_n}{d\mathbf{p}} \right]$$

Backward pass for the adjoints:

$$\left(\frac{\partial \mathbf{R}_n}{\partial \mathbf{x}_n} \right)^{\top} \boldsymbol{\lambda}_n = - \left(\frac{\partial O_n}{\partial \mathbf{x}_n} \right)^{\top} - \left(\frac{\partial \mathbf{R}_{n+1}}{\partial \mathbf{x}_n} \right)^{\top} \boldsymbol{\lambda}_{n+1}, \quad n = N, N-1, \dots, 1$$

$$\frac{dJ_{\lambda}}{d\mathbf{p}} = \sum_{n=1}^N \left(\frac{\partial \mathbf{R}_n}{\partial \mathbf{p}} \right)^{\top} \boldsymbol{\lambda}_n$$

- With $\mathbf{R} : \mathbb{R}^m \rightarrow \mathbb{R}^m$, the Jacobian is $\mathbb{R}^{m \times m}$ (machine learning $\mathbf{R} : \mathbb{R}^m \rightarrow \mathbb{R}^k$, $k \ll m$)
 - Generally, not simple to decide between forward and backward AD
 - However, we are saved by *sparsity* (but it can be difficult to utilize)
- In practice, we only solve $\mathbf{R}_n(\mathbf{x}_n, \mathbf{x}_{n-1}, \mathbf{p}, \mathbf{q})$ to a prescribed tolerance:
 - Accuracy of the computed gradient depends highly on this tolerance
 - Errors may accumulate
- To solve the adjoint equations, all \mathbf{x}_n must be available
 - Typically write and read from disk
 - Special strategies for very large models
- The computed matrices $\partial \mathbf{R}_n / \partial \mathbf{x}_n$ are not necessarily the correct Jacobians
 - Typically computed using the same code as the forward solution
 - This code may contain logic, specialized techniques and manipulations that invalidates AD



Complex multiphysics workflow realized by SLIM group (Georgia Tech, Prof. F. Herrmann) using the Jutul flow solver as one out of several components

Minimize over latent variable z

$$\frac{1}{2} \|\mathcal{F} \circ \mathcal{R} \circ \mathcal{S}(\mathcal{G}_{\theta^*}(\mathbf{z})) - \mathbf{d}\|_2^2 + \frac{\lambda}{2} \|\mathbf{z}\|_2^2$$

where

\mathbf{d} : time-lapse seismic data

\mathcal{G}_{θ^*} : maps latent variable to permeability \mathbf{K}

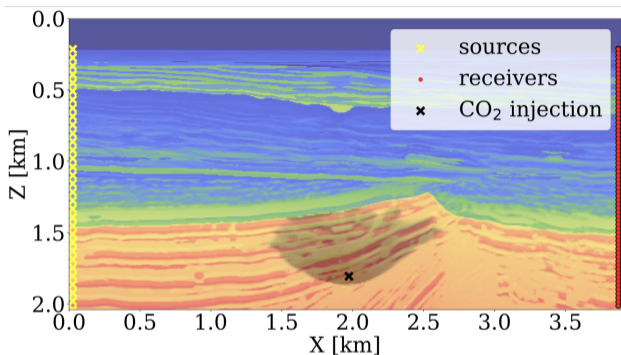
\mathcal{S} : flow physics (Jutul)

\mathcal{R} : rock physics

\mathcal{F} : wave physics module

AD + adjoints provide accurate Jacobians of \mathcal{S} with respect to \mathbf{K}

This ensures interoperability with other packages in Julia's AD ecosystem



Minimize over latent variable z

$$\frac{1}{2} \|\mathcal{F} \circ \mathcal{R} \circ \mathcal{S}(\mathcal{G}_{\theta^*}(z)) - \mathbf{d}\|_2^2 + \frac{\lambda}{2} \|z\|_2^2$$

where

\mathbf{d} : time-lapse seismic data

\mathcal{G}_{θ^*} : maps latent variable to permeability \mathbf{K}

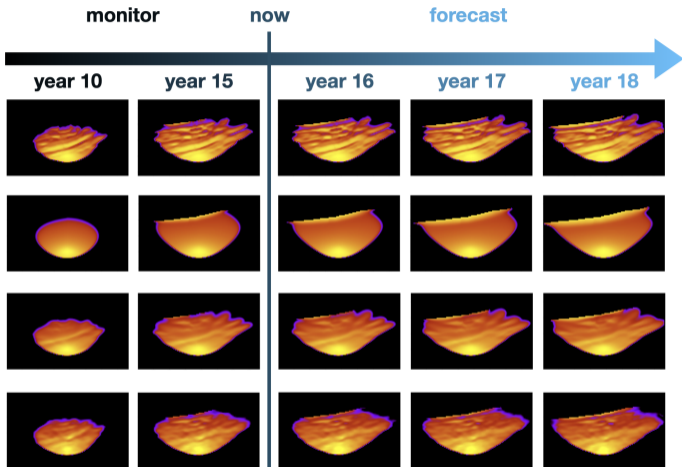
\mathcal{S} : flow physics (Jutul)

\mathcal{R} : rock physics

\mathcal{F} : wave physics module

AD + adjoints provide accurate Jacobians of \mathcal{S} with respect to \mathbf{K}

This ensures interoperability with other packages in Julia's AD ecosystem



Minimize over latent variable z

$$\frac{1}{2} \|\mathcal{F} \circ \mathcal{R} \circ \mathcal{S}(\mathcal{G}_{\theta^*}(z)) - \mathbf{d}\|_2^2 + \frac{\lambda}{2} \|z\|_2^2$$

where

\mathbf{d} : time-lapse seismic data

\mathcal{G}_{θ^*} : maps latent variable to permeability \mathbf{K}

\mathcal{S} : flow physics (Jutul)

\mathcal{R} : rock physics

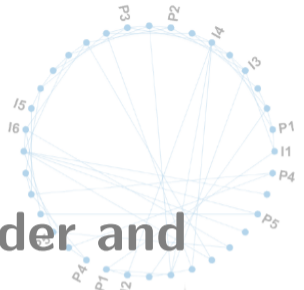
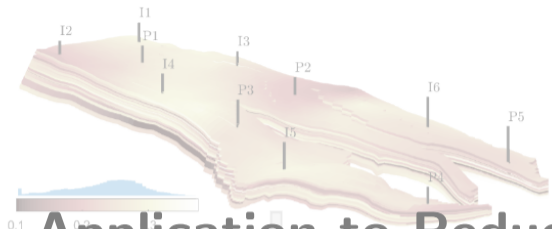
\mathcal{F} : wave physics module

AD + adjoints provide accurate Jacobians of \mathcal{S} with respect to \mathbf{K}

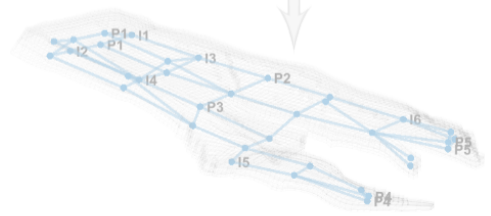
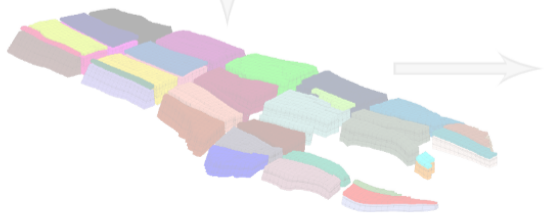
This ensures interoperability with other packages in Julia's AD ecosystem

Making your simulator differentiable and open source makes it more versatile and simplifies inclusion into other workflows

From: M. Louboutin et al. Learned multiphysics inversion with differentiable programming and machine learning. arXiv:2304.05592 [cs.MS]

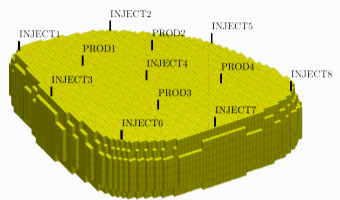


Application to Reduced-Order and Data-Driven Modelling

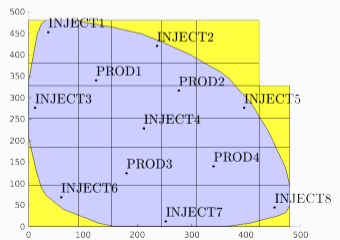
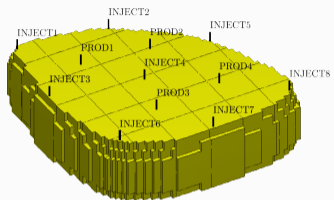


Why use data-driven or reduced-order models?

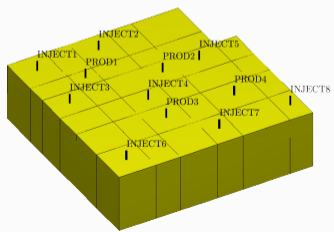
- Traditional reservoir simulation models take long time to build
- Forward simulations are computationally costly
- Production optimization: may require hundreds of model evaluations
- Digital twins: desire for models that can be “continuously” updated with incoming data



Given fine model: agglomerate blocks



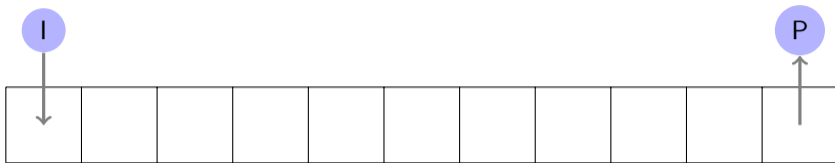
Given map outline: fit Cartesian blocks



- Observations: more connections and parameters seem to improve training ability
- Why not use coarse grid instead of a “streamtube” approach?
- Advantages:
 - more parameters, fewer grid cells
 - graph topology does not depend on well placement

References:

- CGNet: Lie & Krogstad, Geoenery Science and Engineering, 2023. DOI: 10.1016/j.petrol.2022.111266
- TriNet: Devold, MSc thesis, NTNU, 2023
- Krogstad, MS26, Tue 10:05, Dræggen 7



Continuous

$$\frac{\partial}{\partial t} (\phi \rho_\alpha S_\alpha) + \nabla \cdot (\rho_\alpha \vec{v}_\alpha) = q_\alpha$$

$$\vec{v}_\alpha = -\mathbf{K} \lambda_\alpha (\nabla p - \rho_\alpha g \nabla z)$$

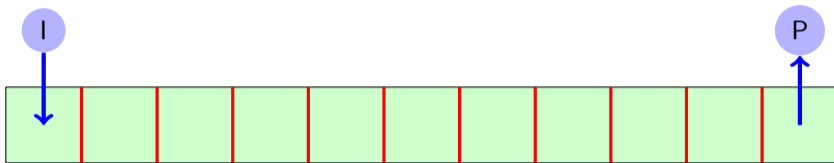
$$q_\alpha = \lambda_\alpha^{wb} \mathbf{J}(p^{wb} - p)$$

Discrete

$$\frac{1}{\Delta t} [(\mathbf{S}_\alpha \rho_\alpha)^{n+1} - (\Phi \mathbf{S}_\alpha \rho_\alpha)^n] + \text{div}(\rho_\alpha \mathbf{v}_\alpha^{n+1}) = \mathbf{q}_\alpha$$

$$\mathbf{v}_\alpha = -\mathbf{T} \text{upw}(\lambda_\alpha) (\text{grad}(p) - g \text{avg}(\rho_\alpha) \text{grad}(z))$$

$$\mathbf{q}_\alpha = \lambda_\alpha^{wb} \mathbf{J}(p^{wb} - p)$$



Continuous

$$\frac{\partial}{\partial t} (\phi \rho_\alpha S_\alpha) + \nabla \cdot (\rho_\alpha \vec{v}_\alpha) = q_\alpha$$

$$\vec{v}_\alpha = -\mathbf{K} \lambda_\alpha (\nabla p - \rho_\alpha g \nabla z)$$

$$q_\alpha = \lambda_\alpha^{wb} \mathbf{J}(p^{wb} - p)$$

Discrete

$$\frac{1}{\Delta t} [(\Phi \mathbf{S}_\alpha \rho_\alpha)^{n+1} - (\Phi \mathbf{S}_\alpha \rho_\alpha)^n] + \text{div}(\rho_\alpha \mathbf{v}_\alpha^{n+1}) = \mathbf{q}_\alpha$$

$$\mathbf{v}_\alpha = -\mathbf{T} \text{upw}(\lambda_\alpha) (\text{grad}(p) - g \text{avg}(\rho_\alpha) \text{grad}(z))$$

$$\mathbf{q}_\alpha = \lambda_\alpha^{wb} \mathbf{J}(p^{wb} - p)$$

Misfit function:

$$M(\boldsymbol{\theta}) = \mathbf{m}(\boldsymbol{\theta})^T \mathbf{m}(\boldsymbol{\theta})$$

Here,

\mathbf{m} – scaled misfit, $m_i = (y_i(\boldsymbol{\theta}) - y_i)/w_i$

y_i – the i th data point (flow rate, bhp, etc.)

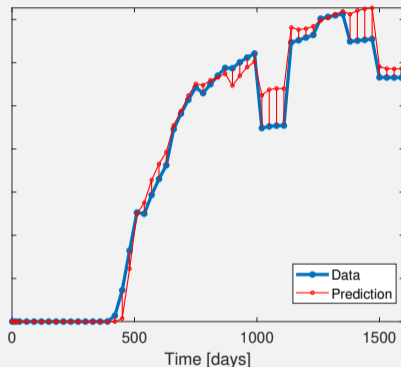
$y_i(\boldsymbol{\theta})$ – data point predicted by GPSNet

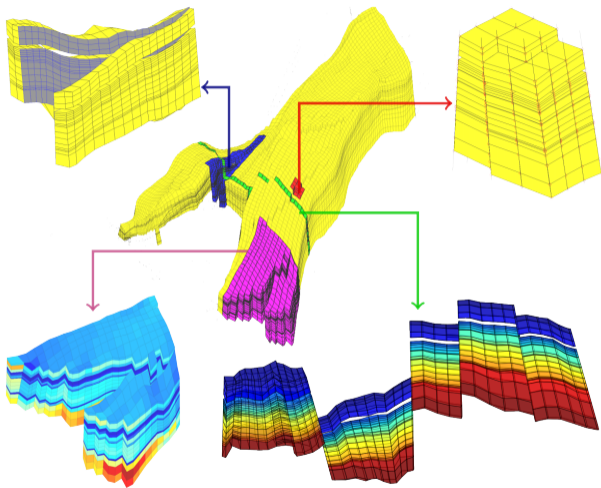
w_i – weight for data point i

$\boldsymbol{\theta}$ – tunable parameters, $\{V, T, J\}$

In addition, we impose certain physical bounds on $\boldsymbol{\theta}$

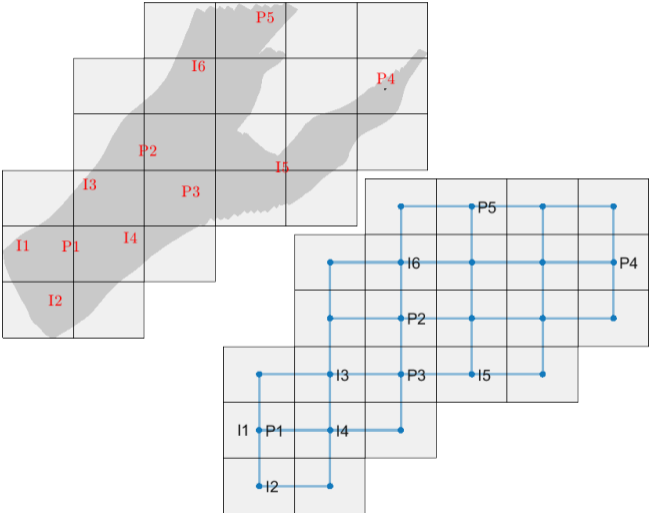
Method: Gauss–Newton with damping (Levenberg–Marquardt),
Jacobians computed from adjoint equations





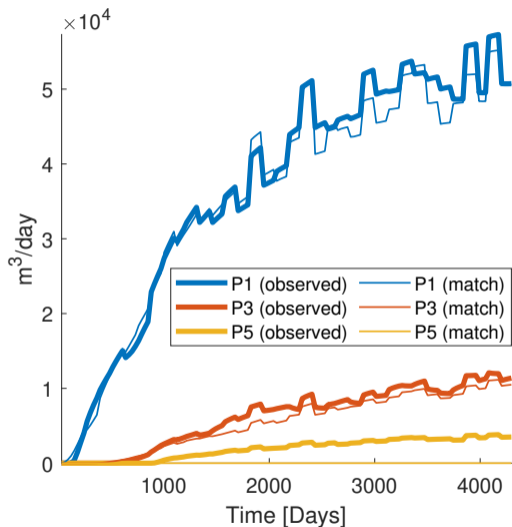
Semi-realistic model:

- Real-field grid:
github.com/OPM/opm-data
- $46 \times 112 \times 22$, 44 915 active cells
- Complex grid: faults, erosions, inactive cells, etc.
- Permeability/porosity: from Lorentzen et al. (SPE J., 2019)
- oil–water system, quadratic relperm, viscosity ratio 5:1
- initial state: filled with oil
- six injectors, constant rate
- five producers, constant bhp



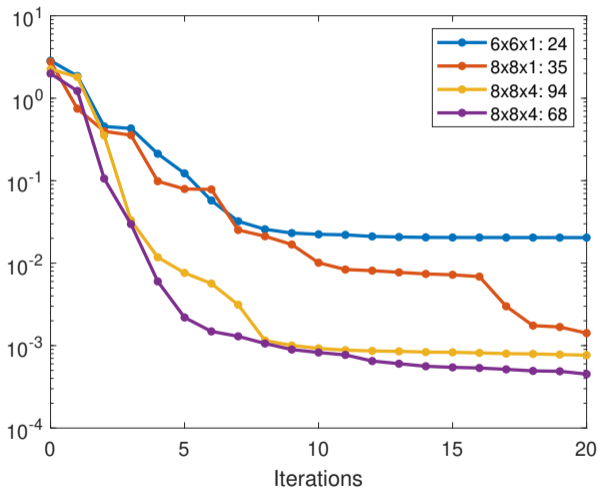
CGNet:

- Fit $6 \times 6 \times 1$ mesh to map outline
- Potential problems:
 - top/bottom surfaces not represented
 - wells I1 and P1 within same cell



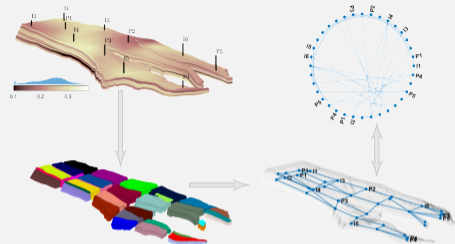
CGNet:

- Fit $6 \times 6 \times 1$ mesh to map outline
- Potential problems:
 - top/bottom surfaces not represented
 - wells I1 and P1 within same cell
- Fit to data is nonetheless ok, except in producer P5 (initially poorly connected and hence sensitivity ≈ 0)

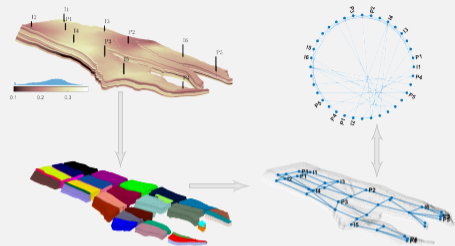
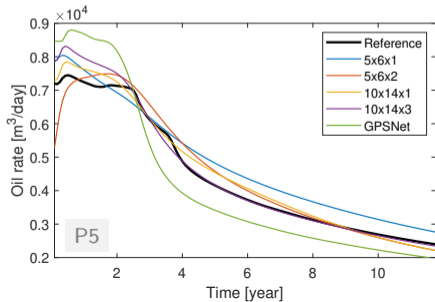
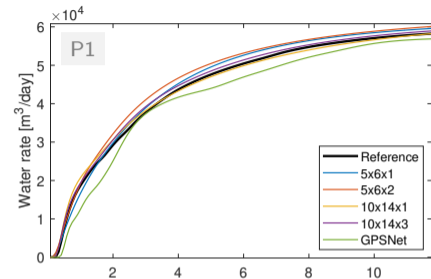


CGNet:

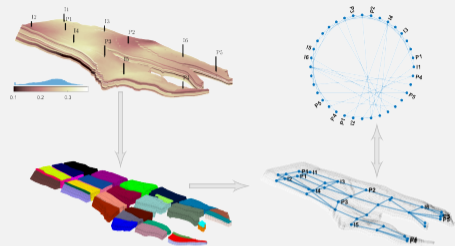
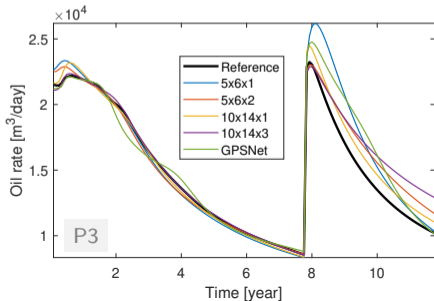
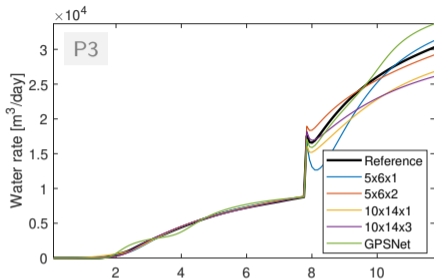
- Fit $6 \times 6 \times 1$ mesh to map outline
- Potential problems:
 - top/bottom surfaces not represented
 - wells I1 and P1 within same cell
- Fit to data is nonetheless ok, except in producer P5 (initially poorly connected and hence sensitivity ≈ 0)
- Increase to 8×8 to avoid multiple wells within the same cell
- Add layers to also represent top and bottom surfaces
- Cull some partially overlapping cells



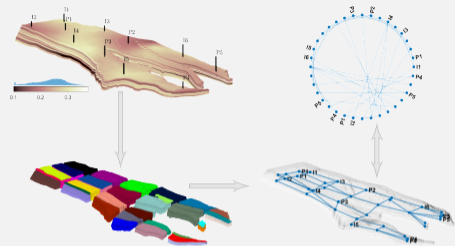
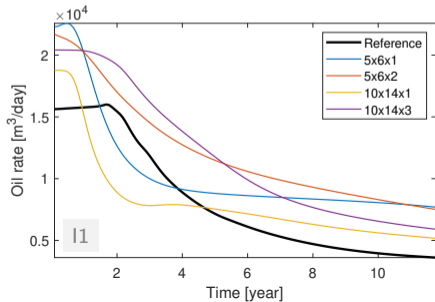
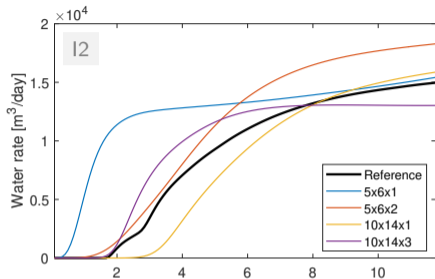
- Rectangular partition in index space + split disconnected blocks
- Training data: random variation around prescribed bhp and rate controls
- Motivation: excite more reservoir states in training data



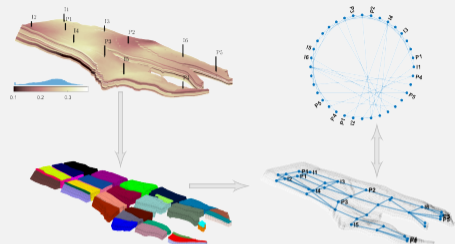
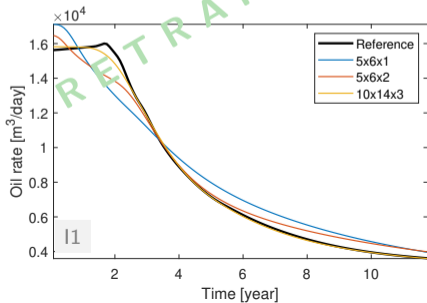
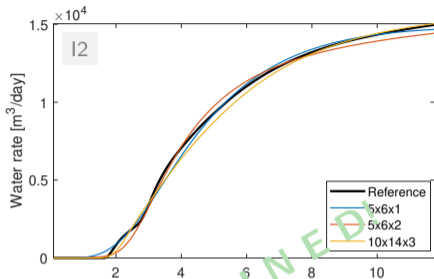
- Rectangular partition in index space + split disconnected blocks
- Training data: random variation around prescribed bhp and rate controls
- Motivation: excite more reservoir states in training data
- **Case 1: Fixed but different bhp/rates**



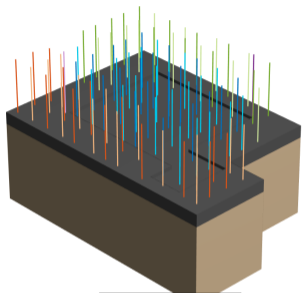
- Rectangular partition in index space + split disconnected blocks
- Training data: random variation around prescribed bhp and rate controls
- Motivation: excite more reservoir states in training data
- Case 2: P1 shut-in after 8 yrs



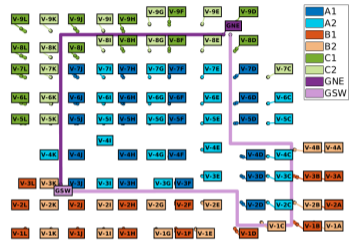
- Rectangular partition in index space + split disconnected blocks
- Training data: random variation around prescribed bhp and rate controls
- Motivation: excite more reservoir states in training data
- **Case 3: I1/I2 converted to producers, P1 shut-in**



- Rectangular partition in index space + split disconnected blocks
- Training data: random variation around prescribed bhp and rate controls
- Motivation: excite more reservoir states in training data
- **Case 3: I1/I2 converted to producers, P1 shut-in**



Gravel layer and reservoir

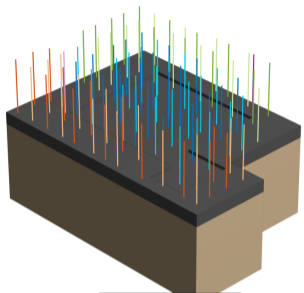


Wells (from above)

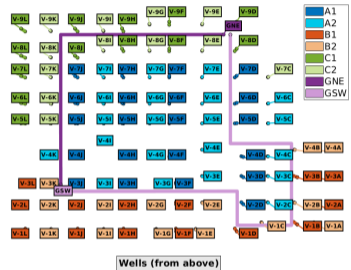
- Residential/commercial building complex in Asker
- Complex multi-reservoir geothermal storage facility: three reservoirs, one hundred wells
- Covers heating and cooling needs
- Also provides energy to snow-melting in city streets



Residential/commercial building complex in Asker, Norway
wesselkvartalet.no

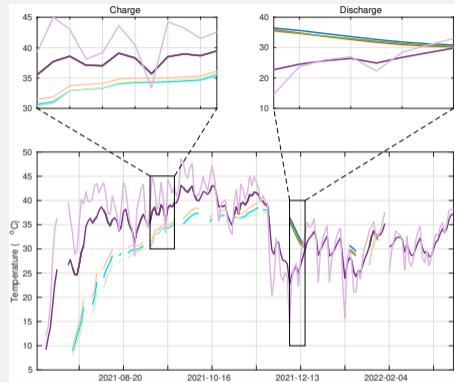


Gravel layer and reservoir



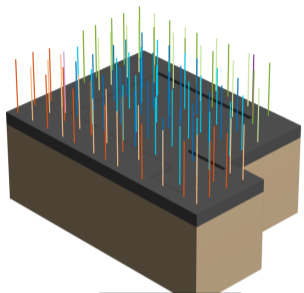
Wells (from above)

- Residential/commercial building complex in Asker
- Complex multi-reservoir geothermal storage facility: three reservoirs, one hundred wells
- Covers heating and cooling needs
- Also provides energy to snow-melting in city streets

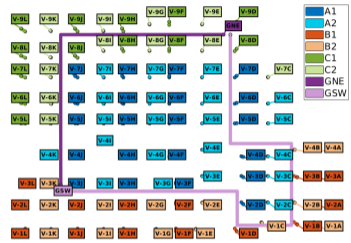


Simulation of the entire history using observed injection temperatures

Klemetsdal/Andersen, MS65, Thu 14:35/15:00

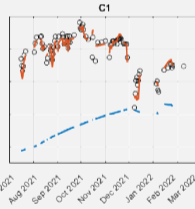
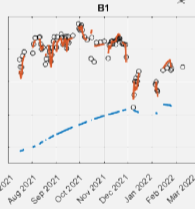
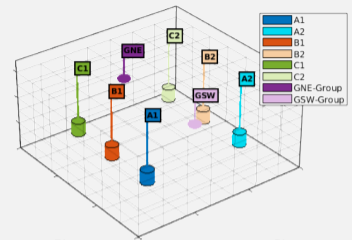


Gravel layer and reservoir



Wells (from above)

- Residential/commercial building complex in Asker
- Complex multi-reservoir geothermal storage facility: three reservoirs, one hundred wells
- Covers heating and cooling needs
- Also provides energy to snow-melting in city streets



CGNet with 294 reservoir nodes tuned to match manifold temperature, results after 50 quasi-Newton iterations

Tried to make a case for the **importance** of differentiable simulators and the **utility** of automatic differentiation

- First part discussed how to apply AD to **dynamic variables** to aid simulator development:
 - avoid hand-calculation and explicit implementation of Jacobians
 - aid in developing fully coupled multiphysics simulators
- Second part showed how to apply AD to **parameters** in workflows:
 - use of differentiable simulator to generate data-driven or reduced-order models
 - could also have demonstrated optimization workflows
- However, none of the simulators are **fully differentiable**:
 - differentiable with respect to parameters in the continuous equations
 - not differentiable to grid, tolerances and program control parameters, etc.
 - generally challenging to know upfront where the need for gradients will pop up